

플래시 메모리 상에서의 효율적인 동작을 위한 수정 B-트리 인덱스

노홍찬⁰ 김승우 김우철 박상현
연세대학교 컴퓨터과학과

{fallsma⁰, kimsuw, twelvepp, sanghyun}@cs.yonsei.ac.kr

Modified B-Tree Index for Efficiency on the Flash-Memory Storage System

Hongchan Roh⁰ Seungwoo Kim Woo-Cheol Kim Sanghyun Park
Department of Computer Science, Yonsei University

요 약

플래시 메모리는 기술 발전에 따른 빠른 용량 증가와 모바일 환경에 우수한 특성으로 인해 가까운 시일 내에 하드 디스크를 대체할 대용량 저장 장치로서 주목 받고 있다. 이러한 흐름에 따라 플래시 메모리 사용이 증가하고 플래시 메모리에 저장하는 데이터의 양이 점차 많아지면서 플래시 메모리 상의 효율적인 인덱스 구조에 대한 필요성도 함께 증가하고 있다. 하지만 기존의 대표적인 인덱스 중 하나인 B-트리 인덱스를 플래시 메모리에 적용하기 위해서는 하드 디스크와 플래시 메모리 간의 다른 특성들 때문에 플래시 메모리에 맞게 인덱스 구조를 수정하는 작업이 필요하다. 본 연구에서는 이를 해결하기 위한 기존의 연구에 대해 소개하고 기존 연구의 한계점과 이를 개선한 인덱스 구조를 제안한다.

1. 서론

플래시 메모리는 최근 하드 디스크를 대체할 수 있는 대용량 저장 장치로 인정받고 있다. 하드 디스크와 유사하게 비휘발성의 특징은 유지하면서도 하드 디스크보다 충격에 강하고 전력 손실이 작은 장점이 있다. 플래시 메모리 설계 기술의 발전으로 플래시 메모리의 용량이 빠른 속도로 증가하여 이러한 증가 속도대로 라면 머지않아 하드 디스크의 용량을 넘어설 것으로 기대된다.

플래시 메모리의 종류는 크게 낸드(NAND) 플래시 메모리와 노어(NOR) 플래시 메모리가 있으며 낸드 플래시가 저장 장치를 구성하는데 우수한 성질을 가진다[8].

플래시 메모리에서 동작하는 파일 시스템은 크게 두 가지 분류로 나뉘어진다. 고유파일시스템(native file system) 방법과 블록장치모방(block-device emulation) 방법이다[6]. 블록장치모방 방법은 기존의 디스크 기반의 파일 시스템을 모방(emulate)하기 때문에 고유파일시스템 방법에 비해 디스크 기반의 작업들을 빠르게 플래시 메모리에 적용시킬 수 있는 장점이 있다. 이러한 블록장치모방 방법을 구현한 시스템 중 FTL[1,2,3] 은 일반적으로 많이 사용하는 방법 중 하나이다[6].

플래시 메모리가 많이 사용되고 플래시 메모리의 용량이 증가할수록 더 많은 양의 데이터를 관리하는 인덱스 구조의 필요성도 함께 증가하고 있다. 이러한 필요성에 따라 위에서 언급한 낸드 플래시 메모리의 FTL상에서 동작하는 인덱스 구조에 대해 생각해볼 수 있다.

기존의 디스크 기반의 다양한 인덱스 기법 중 B-트리 인덱스[9]는 뛰어난 확장성과 효율성으로 일반적으로 많

이 사용하는 인덱스 중 하나이다. 하지만 B-트리를 그대로 플래시 메모리에 적용시키면 플래시 메모리와 하드 디스크의 서로 다른 특성들로 인해 문제가 발생한다.

본 논문은 B-트리를 플래시 메모리의 FTL상에 적용시킬 때 발생하는 문제를 알아보고 이 문제를 해결하고자 한 기존 연구의 문제점을 종합하여 새로이 문제를 정의하고 그에 대한 해결 방법을 제시한다.

2장에서는 FTL을 중심으로 플래시 메모리의 특징들에 대해 좀 더 자세히 소개하고 B-트리를 FTL상에 적용하는 문제와 관련 연구에 대해 다룬다. 3장에서는 이 논문에서 해결하고자 하는 문제를 정의하고 그에 대한 해결 방안을 제시한다. 4장에서는 이 논문의 해결 방안을 기존 연구와 이론적으로 비교한 성능 분석 결과를 제시하고 5장에서는 본 논문의 결론과 앞으로의 연구 계획을 설명한다.

2. 관련 연구

2.1 플래시 메모리의 특징

낸드 플래시 메모리는 큰블록(large block) 방식과 작은블록(small block) 방식으로 나뉘지만 두 가지 방식은 구조와 성능에서 조금 차이가 날 뿐 기본적인 구조와 특징을 공유한다.

낸드 플래시의 기본적인 구조는 최소 단위의 저장공간인 페이지(page)와 여러 페이지가 모인 블록(block)으로 구성된다. 기본적인 특징으로는 1) 첫 번째로 읽기, 쓰기 작업은 페이지 단위로 수행되지만 지우기 작업은 블록 단위로 수행된다는 것을 들 수 있다. 2) 두 번째로 페이지에 대한 읽기 속도와 쓰기 속도, 블록에 대한 지우기 속도가 각각 다르다는 점을 들 수 있다. 일반적으로 페이지에 대한 쓰기 속도가 읽기 속도에 비해 10배 정도

본 논문은 서울특별시가 추진 중인 2005년도 '서울시 산학연 협력사업(10660)'지원의 일환으로 작성되었음.

표 1. 1기가바이트 큰블록 낸드 플래시 메모리 특징¹⁾

페이지 크기	블록 크기	페이지 읽기 속도	페이지 쓰기 속도	블록 지우기 속도
2kbyte	128kbyte	20 μ s	200 μ s	1.5ms

느리며 블록에 대한 지우기 속도가 페이지에 대한 쓰기 속도에 비해 10배 정도 느리다.

큰블록 방식의 플래시 메모리의 예로 1기가바이트(gigabyte) 큰블록 낸드 플래시 메모리의 구조와 성능을 표 1에 정리하였다.

위에서 언급한대로 낸드 플래시 메모리의 읽기, 쓰기 작업의 단위가 페이지이고 지우기의 단위는 블록이므로 페이지 단위의 수정이 발생할 때 해당 페이지만 덮어쓰지 못하고 전체 블록을 지워야만 해당 페이지의 데이터를 수정할 수 있는 문제점이 발생한다. 이러한 문제점을 해결하기 위해 최대한 지우는 작업을 뒤로 미루고 수정이 필요한 페이지에 대해서는 해당 페이지의 내용을 아직 쓰여 지지 않은 다른 페이지에 복사해야 한다. 이러한 수정 작업을 외부공간갱신(out-place update)이라 한다. 그리고 아직 쓰여 지지 않은 페이지 즉, 플래시 메모리가 동작 후 한 번도 쓰여 지지 않았거나 해당 페이지에 대해 지우기 작업이 수행된 후 쓰여 지지 않은 페이지를 프리페이지(free page)라 한다. 외부공간갱신이 수행된 후 올라르지 못한 데이터를 가지고 있는 기존의 페이지를 데드페이지(dead page)라 하고 기존 페이지의 데이터가 복사된 페이지는 라이브페이지(live page)라 한다 [6,8].

수정 작업의 결과로 외부의 접근에 대해 항상 올바른 데이터를 연결해줄 수 있는 방법이 필요하게 된다. 이러한 필요성으로 가상 주소를 만들어 해당 가상 주소가 항상 올바른 데이터를 가리키도록 한다. 이러한 가상 주소를 LBA(Logical Block Address)라 하고 LBA를 실제 플래시 메모리 상의 물리적 주소와 맵핑(mapping)할 수 있는 중계테이블(translation table)을 가진다[6].

시간이 지나면 데드페이지가 플래시 메모리에 계속 쌓이게 되고 데드페이지에서 수정된 데이터를 복사할 프리페이지가 남지 않게 된다. 이렇게 되면 플래시 메모리의 데이터에 대한 더 이상의 수정이나 추가가 불가능하다. 그러므로 이러한 데드페이지를 프리페이지로 바꿔줄 수 있는 작업이 필요하다.

가비지컬렉션(garbage collection)은 특정 블록을 선택하고 그 블록을 지움으로써 프리페이지를 생산해내는 작업이다. 여기서 중요한 것은 블록 선택의 기준이다. 어떤 블록을 선택해서 그 블록을 지우고 블록 내의 페이지들을 프리페이지로 생성해낼 것인가 하는 문제이다. 이러한 블록 선택의 기준을 블록재생정책(block-recycling policy)이라고 한다[6,8].

블록재생정책에 관한 연구로서 Kawaguchi 등[4]은 cost-benefit policy를 제안하였다. 이는 해당 블록의 데드 페이지의 개수와 최근 사용 후 지난 시간을 기준으로 휴리스틱 함수를 정의하고 해당 함수의 값이 가장 큰 블록

을 선택하여 가비지컬렉션을 수행하는 하는 방법이다.

또 다른 플래시 메모리의 중요한 특징은 한 블록을 지울 수 있는 횟수가 최대 만 번에서 백만 번 사이로 제한되어 있다는 것이다[8]. 이를 해결하기 위해 지우는 작업이 한 블록에만 몰리지 않고 전체 플래시 메모리 블록에 골고루 분배될 수 있도록 하는 작업이 필요하고 이를 웨어레벨링(wear-leveling)이라 한다[8].

이와 관련되어 Kwoun 등[5]은 주기적으로 라이브페이지의 데이터들을 블록 간에 옮기는 방법을 제안하였다. 라이브페이지의 데이터를 다른 블록으로 옮김으로써 전체 블록의 지워진 횟수를 비슷하게 유지하는 방법이다.

이러한 플래시 메모리 시스템의 중요한 특징인 가비지컬렉션과 웨어레벨링은 플래시 메모리에 데이터를 쓰는 작업과 밀접하게 연관되어 발생된다. 그러므로 플래시 메모리에 대한 쓰기 작업은 쓰기 작업 자체에 의한 비용 외에 가비지컬렉션과 웨어레벨링에 의한 추가 비용을 초래한다.

2.2 B-트리를 플래시 메모리에 적용시키는 문제

기존 B-트리를 그대로 플래시 메모리에 적용했을 경우 생기는 문제점은 최악의 경우 분할(split)이 발생하기 전까지 하나의 입력 작업(insert) 마다 하나의 리프노드(leaf node)를 수정해야 한다는 점이다. 그렇게 되면 리프노드의 크기만큼의 플래시 메모리 공간에 쓰기 작업을 수행해야 하고 한 리프노드가 m개의 레코드로 구성된다면 한 리프노드를 구성하기 위해 m개의 입력에 대해서 매번 해당 리프노드의 공간에 대한 쓰기 작업을 수행해야 한다. 이러한 쓰기 작업은 단위 쓰기 작업의 속도가 느리므로 큰 비용을 가지게 된다. 또한 플래시 메모리의 외부공간갱신 특징으로 인해 리프노드 공간의 m배에 해당하는 영역에 동일한 리프노드의 데이터들이 저장된다. 하나의 노드가 이렇게 많은 공간을 차지하면 전체의 노드는 더 많은 공간을 차지하게 되고 프리페이지가 빠르게 소모된다. 이로 인해 가비지컬렉션과 웨어레벨링이 자주 일어나게 된다. 이렇듯 기존 B-트리를 그대로 적용했을 경우 B-트리 인덱스에 대한 입력 작업으로 인한 매우 큰 쓰기 비용이 발생하게 된다. 이러한 이유로 쓰기 비용을 줄일 수 있는 수정된 B-트리 구조가 필요하다.

B-트리 쓰기 비용을 감축하기 위한 기존 연구로서 Chin-Hsien 등[6]이 제안한 BFTL이 있다. BFTL은 쓰기 비용을 줄이기 위해 B-트리를 위한 계층(layer)을 구성하는 아이디어를 사용했으며 계층을 구현하기 위한 방법으로 노드전환테이블(node transition table)과 예약버퍼(reservation buffer)를 제안했다. 이러한 구조를 통해 BFTL은 FTL이 디스크 기반 파일 시스템을 모방한 것처럼 가상 노드를 구성하여 한 노드가 한 페이지로 구성된 B-트리와 같이 작동한다. BFTL은 FTL을 기반으로 구축되어 BFTL에서의 입출력 작업은 모두 FTL의 LBA상에서

1) http://www.samsung.com/Products/Semiconductor/common/product_list.aspx?family_cd=NFL0201

이루어진다.

BFTL에서 예약버퍼의 역할은 입력으로 들어온 엔트리(entry)²⁾들을 모았다가 현재의 가상 노드들의 경계로 엔트리들을 묶고 해당 묶음을 플래시 메모리의 저장 공간을 최소로 차지하도록 FTL의 LBA 위에 쓰는 것이다. 이렇게 가상 노드별 엔트리들의 묶음으로 쓰기 작업을 수행하게 되면 여러 번의 입력 작업 당 하나의 리프노드를 쓰게 되는 효과를 얻을 수 있다.

BFTL의 노드전환테이블은 이렇게 예약버퍼로부터 쓰여진 엔트리들을 가상 노드별로 모으는 역할을 한다. 엔트리들은 현재의 예약버퍼상에서 각 노드별로 묶음을 이뤄 LBA상에 쓰여 지고 한번 쓰여질 때 가상 노드의 모든 엔트리들이 한 묶음으로 포함되기 어려우므로 한 가상 노드의 엔트리들은 LBA상에 여러 조각으로 분산된다. 노드전환테이블은 이러한 조각들을 해당 가상 노드별로 모아 그 리스트를 각 가상 노드별로 메모리에 유지한다.

노드전환테이블의 크기가 커지면 검색에 있어 큰 문제가 발생한다. 가상 노드 하나가 LBA상에 n 개의 조각으로 구성되어 n 개의 페이지에 흩어진다면 해당 노드 하나를 검색하기 위해서 n 개의 페이지를 읽어야 하는 비용이 들 수 있기 때문이다. 이러한 문제로 BFTL은 노드전환테이블에서 하나의 가상 노드를 구성하는 조각의 개수를 임계값 C 로 제한하고 C 이상의 조각이 구성될 때 해당 조각들을 모아 가상의 노드와 일치하는 LBA 상의 노드로 쓰는 컴팩션(compact)을 수행한다. 이렇게 하여 노드전환테이블의 가상 노드의 조각 리스트를 C 이하의 개수로 유지한다. 이는 가상 노드에 속하는 조각들을 모아 다시 LBA 상에 써야하는 추가적인 쓰기 비용을 발생시킨다.

이러한 BFTL의 문제점은 다음과 같다. 첫째, 검색 작업 시에 노드전환테이블에서 관리하는 가상 노드의 모든 리스트를 읽어야 하는 단점이 있다. 리스트의 길이를 C 로 제한하므로 최악의 경우 BFTL은 가상 노드 하나당 C 번의 읽기 작업이 필요하게 되고 B-트리의 검색 작업(search)으로 인한 읽기 비용의 C 배에 해당하는 읽기 비용을 가지게 된다. 둘째, 가상 노드를 구성하는 조각의 개수를 C 개 이하로 유지하기 위해 주기적인 컴팩션을 수행하고 그로 인한 추가 쓰기 비용이 발생하기 때문에 예약버퍼에 의해 절약되는 쓰기 비용을 상쇄시킨다. 셋째, 하나의 계층을 더 뚫으로써 확장성과 신뢰성이 떨어진다. 기존 B-트리의 노드 사이즈가 1페이지인 특화된 구조에 의지하므로 차후에 노드 사이즈를 늘린다던지 인덱스 구조를 변경할 수 있는 확장성이 부족하며 메모리에 많이 의존함으로써 외부의 충격으로 데이터를 잃어버릴 수 있다.

이러한 BFTL의 한계점을 해결하면서 플래시 메모리에서 B-트리의 쓰기 비용을 줄일 수 있는 방법이 필요하다.

3. MB-트리의 인덱싱 기법

3장에서는 2장에서 알아본 문제점들을 토대로 플래시 메모리에 B-트리를 적용하기 위한 문제를 정의하고 본

논문에서 제안하는 인덱싱 기법인 수정 B-트리 인덱스(MB-트리)를 설명한다.

3.1 문제 정의

2장에서 알아본 문제점들을 토대로 정리해보면 플래시 메모리에 B-트리를 적용하기 위해서는 고려해야 할 사항은 다음과 같다.

첫째, 인덱스에 대한 입력 작업으로 인한 쓰기 비용을 줄일 수 있는 B-트리 구조가 필요하다.

둘째, 쓰기 비용을 개선하면서도 BFTL보다 검색 작업으로 인한 읽기 비용이 작아야 한다.

셋째, BFTL처럼 특정 계층을 두어 B-트리를 모방하지 않고 인덱스 구조 자체의 수정을 통해 통합된 하나의 인덱스 구조를 이루어야 한다.

3.2 MB-트리 개요

제안하는 MB-트리의 개략적인 구조를 그림 1과 그림 2에 표시하였다. MB-트리는 출력버퍼, 중간 노드(internal node), 리프노드, 리프페이지, 메타페이지(meta page)로 구성된다. 중간 노드는 기존 B-트리와 동일한 구성을 가지며 자식 노드(child node)들의 경계값을 키 값으로 가지고 자식 노드를 포인팅(pointing) 한다. 리프노드는 1개 이상의 리프페이지와 1개의 메타페이지로 구성된다. 리프페이지는 리프노드를 구성하는 단위로서 실제 엔트리들을 포함한다. 메타페이지는 해당 리프노드의 모든 엔트리들의 순서정보를 저장한다. 각각의 구성 방법에 대해서는 3.3절에서 좀 더 자세히 설명하며 다음으로 출력버퍼와 메타페이지의 개념에 대해 간략히 설명한다.

입력 작업에 의한 쓰기 비용을 줄이기 위한 방법으로 출력버퍼 활용을 제안한다. 이는 입력되는 엔트리들을 출력버퍼에 한동안 모았다가 출력버퍼가 다 찼을 때 한꺼번에 여러 개의 엔트리를 쓰는 방법이며 3.3절에서 이러한 출력버퍼의 출력 정책 및 구성에 대해 설명한다.

위의 방법에 따라 리프 노드에 쓰여진 엔트리들은 출력 시에 함께 출력된 묶음 내에서는 정렬된 순서가 유지되지만 리프 노드 내에서는 정렬된 순서가 유지되지 못한다. 이러한 리프 노드에 대해 검색 작업이 수행되면 모든 엔트리들을 읽어야 하는 비용이 발생 하므로 검색 비용을 줄이기 위해 정렬된 순서를 보장할 수 있는 방법이 필요하다. 이러한 방법으로 하나의 리프 노드 당 해당 노드의 순서 정보를 관리하는 메타 페이지의 사용을 제안하며 3.4절에서 그 구성 방법에 대해 설명한다.

3.3 출력버퍼의 출력 정책 및 MB-트리 구조

출력버퍼는 버퍼가 가득차면 한꺼번에 다수의 엔트리를 MB-트리의 해당 리프노드에 쓰게 된다. 이러한 출력버퍼의 작업을 출력 작업이라 정의하고 출력 작업을 다음 다섯 단계로 나누어 설명한다.

첫째, 출력버퍼로 할당된 메모리 공간이 다 찼 때까지 입력을 받아 해당 입력들의 엔트리들을 저장하고 저장된 모든 엔트리들을 정렬하여 유지한다.

둘째, 출력버퍼가 다 차면 현재의 리프노드들의 경계들로 출력버퍼의 엔트리들을 나누어 리프노드별 엔트리

2) 인덱스를 구성하고자 하는 레코드의 키 값과 해당 레코드에 대한 포인터의 쌍

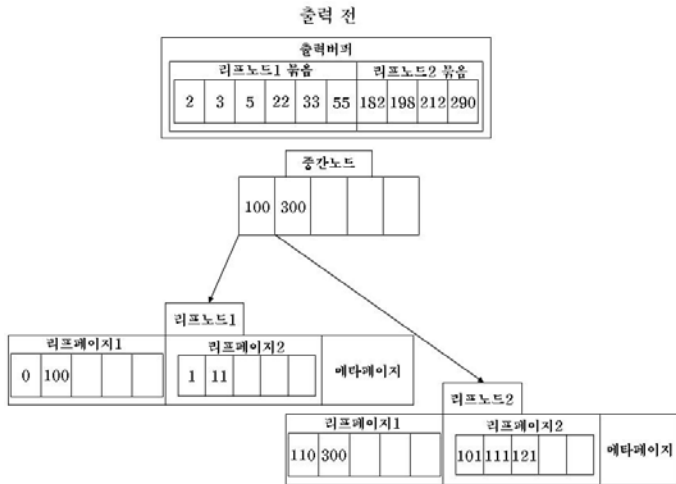


그림 1. MB-트리의 전체 구조 (출력 전)

의 묶음을 만든다.

셋째, 리프노드별 엔트리의 묶음 중 가장 많은 엔트리를 포함하는 묶음을 선택하고 선택된 묶음을 해당 리프노드에 쓰기 위해 다음 정책을 따른다.

넷째, 엔트리의 묶음을 리프노드에 쓰기 위해 쓰여질 엔트리의 묶음의 크기와 리프페이지의 빈 공간 크기를 비교하여 가장 잘 맞는 리프페이지를 선택하는 best fit placement strategy[7]를 사용한다.

다섯째, 네 번째 단계에 의해 선택된 각각의 리프페이지에 해당 리프페이지에 쓰여져야 하는 엔트리들을 추가하고 해당 리프페이지들의 엔트리들을 각각 정렬하여 저장한다.

그림 1, 그림 2의 예제는 출력버퍼가 2페이지, 리프페이지와 중간노드가 1페이지, 1페이지에 들어갈 수 있는 엔트리의 개수가 5개로 구성된 예이다. 그림 1은 출력버퍼에 입력으로 정수 값인 '2, 3, 5, 22, 33, 55, 182, 198, 212, 290'이 들어온 후 출력 작업의 두 번째 단계까지 수행된 상태를 보이고 있다. 입력이 들어온 순서는 정수 값의 순서와 관계없이 랜덤하게 들어왔지만 출력버퍼의 출력 작업 첫 번째 단계를 거치면서 정렬이 되었고 두 번째 단계를 거치면서 각 리프노드의 경계값인 100, 300을 기준으로 각 리프노드별로 나누어져 묶음이 형성되었다.

그림 2는 그림1의 상태에서 출력 작업의 나머지 세 단계가 모두 수행된 상태이다. 세 번째 단계에서 6개의 엔트리를 가지는 리프노드1의 묶음이 선택되어 네 번째, 다섯 번째 단계에 의해 리프노드1의 각 리프페이지에 엔트리들이 3개씩 나뉘어 쓰인 상태이다. 출력버퍼에는 리프노드1의 묶음을 쓰고 난 나머진 리프노드2의 묶음에 해당하는 엔트리들만 남아 있게 되었다.

다음으로 이러한 출력버퍼와 관련하여 MB-트리 구성요소들을 구성하는 방법에 대해 설명한다. 출력버퍼의 크기는 해당 시스템이 허용하는 최대의 메모리 공간으로 구성한다. 출력버퍼가 크면 클수록 많은 엔트리들을 한번에 해당 리프노드에 쓸 수 있기 때문이다.

하나의 리프노드는 하나 이상의 리프페이지로 구성하고 하나의 리프페이지는 해당 파일시스템의 최소 입출력 단위로 구성한다. 이렇게 리프노드를 다수의 리프페이지로 구성하는 이유는 다음과 같다. 출력버퍼가 엔트리들

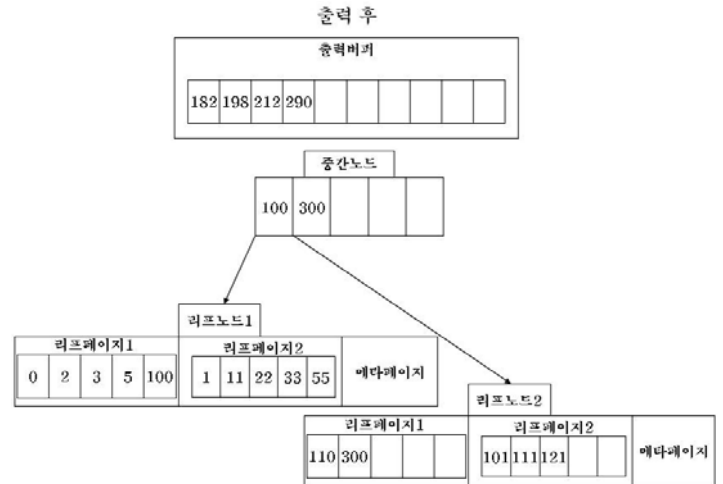


그림 2. MB-트리의 전체 구조 (출력 후)

을 묶음 단위로 쓰므로 한 번에 많은 엔트리가 쓰여지기 때문에 리프노드의 크기(구성하는 리프페이지의 개수)가 작으면 분할이 자주 발생하여 그로 인한 쓰기 비용도 커진다. 또한 리프노드의 크기가 작을수록 동일한 개수의 엔트리들의 입력에 대해 많은 리프노드가 형성된다. 많은 리프노드는 출력버퍼의 출력 작업 시 많은 리프노드의 경계를 형성하고 리프노드별 엔트리의 묶음의 크기를 작게 한다. 묶음의 크기가 작아지면 한 번에 작은 수의 엔트리가 쓰여지게 되어 입력 작업에 대한 쓰기 비용을 증가시킨다.

중간 노드의 경우 최소의 입출력 단위로 구성하고 중간 노드에 입력되는 자식 노드의 경계값의 경우 출력버퍼를 사용하지 않는다. 대신 자식 노드의 분할에 의해 추가되는 경계값을 입력으로 받아 수정된 노드를 바로 플래시 메모리에 쓴다. 중간 노드는 리프노드만큼 수정이 자주 발생하지 않고 중간 노드에 대한 입력은 출력버퍼를 별도로 구성해야 하므로 출력버퍼를 사용하지 않는다. 출력버퍼를 사용하지 않으므로 노드의 크기도 크게 할 필요가 없다.

3.4 메타페이지의 구성과 순서정보 관리 방법

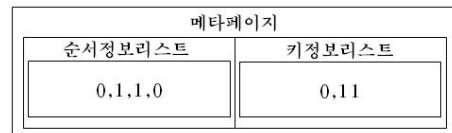


그림 3. 메타페이지 구조

메타페이지는 최소 입출력 단위의 크기를 가지며 해당 리프노드의 모든 엔트리들의 순서정보를 관리한다. 또한 검색의 효율성을 위해 각 순서정보의 구간을 나눈 키정보도 함께 관리한다. 이러한 순서정보와 키정보의 배열(array)을 각각 순서정보리스트, 키정보리스트로 정의하고 메타페이지의 구조를 그림 2에 표시하였다.

순서정보는 리프노드를 구성하는 리프페이지들의 순서번호로 표현된다. 그림 1의 경우 리프노드가 2개의 리프페이지로 구성되므로 왼쪽에서 부터 첫 번째 리프페이지의 경우 순서번호는 1이 되고 두 번째 리프페이지의 순

서번호는 2가 된다. 리프페이지는 3.3절에서 설명한 출력 버퍼의 출력 작업 중 다섯 번째 단계에 의해 항상 정렬되어 있으므로 순서정보를 효율적으로 구성하기 위해 k-way merge[7]의 아이디어를 사용할 수 있다. 리프페이지들 각각을 k-way merge에서 사용하는 run으로 보고 각 run들을 비교하는 과정의 결과를 리프페이지의 순서번호로 기록하여 순서 정보 리스트에 추가한다. 즉, 기록되는 리프페이지의 순서번호는 k-way merge에서 각각의 run의 가장 작은 값을 비교 후 가장 작은 값을 선택하는 과정에서 선택된 가장 작은 값이 있었던 run을 나타내는 것이다. 이렇게 k-way merge의 비교 과정의 결과를 리프페이지의 순서 번호로 표현함으로써 리프페이지의 순서 번호만으로도 전체 엔트리들 간의 순서를 알아낼 수 있도록 한다.

하나의 순서정보는 한정적인 메타페이지의 공간에 최대한 많은 순서정보를 담을 수 있도록 비트 단위로 표현한다. 몇 비트를 설정할 것인지는 한 리프노드를 구성하는 리프페이지의 개수에 의해 결정한다. n개의 리프페이지로 구성된다면 하나의 순서정보는 $\log_2 n$ 비트로 표현된다.

키정보를 구성하는 방법은 다음과 같다. 순서정보리스트를 동일한 순서정보 개수의 구간으로 나누고 해당 구간의 경계값에 해당하는 엔트리의 키(key)값들을 키정보로 구성한다. 구간의 크기(구간을 나누는 순서정보의 개수)는 이 절(3.4) 마지막에서 설명한다. 이렇게 구간을 나누어 경계값을 저장하게 되면 나누어진 구간에서만 이진 검색(binary search)을 수행할 수 있다.

이렇게 정의된 키 정보를 좀 더 압축하여 제한된 메타페이지의 공간 안에 최대한 많이 저장하기 위해 키정보를 압축하여 표현하는 방법을 사용한다. 이를 정규화 작업이라 정의하고 다음과 같은 방법을 따른다. 첫째, 해당 키 값을 표현할 수 있는 최소의 크기를 찾기 위해 리프노드의 경계값 두 개를 가져와 큰 값에서 작은 값을 빼고 이 값을 바이트 단위로 표현할 수 있는 최소의 바이트 수를 찾는다. 둘째, 해당 키 값을 리프노드의 작은 경계값으로 뺀 후 그 결과를 첫 번째 단계에서 계산한 바이트 수로 표현한다.

이렇게 키정보를 구성하고 정규화 작업을 통해 압축하여 배열로 구성한 것을 키정보리스트로 정의하고 이를 메타페이지에 순서정보리스트를 저장하고 남은 공간에 저장한다.

이렇게 계산된 키정보리스트의 공간으로 부터 저장 가능한 키정보의 개수를 계산하고 그로부터 앞에서 언급한 순서정보리스트를 나누는 구간의 크기를 계산한다

그림 2의 메타페이지 구성은 그림 1의 리프노드1에 속하는 메타페이지의 구성을 표현하였다. 순서정보 구성 방법에 따라 각 순서정보의 크기는 리프노드의 리프페이지의 개수가 2개이므로 1비트로 구성한다. 리프노드1에 속하는 엔트리의 키 값인 '0, 1, 11, 100'이 속하는 리프페이지의 번호인 '1, 2, 2, 1'을 순서대로 사용하여 리프페이지 번호가 1인 경우는 0으로 리프페이지 번호가 2인 경우는 1로 표시하여 각 1bit의 '0, 1, 1, 0'의 순서정보리스트를 구성한다. 키정보리스트는 해당 순서정보리스트의 순서정보 개수(4개)의 절반에 해당하는 개수(2개)로

구성하여 첫 번째 순서정보와 세 번째 순서정보에 해당하는 키 값인 '0, 11'을 키정보리스트로 구성하는데 사용한다. 정규화 작업 첫 번째에 따라 각 키정보의 크기는 리프노드1의 경계값이 0, 100이므로 100에서 0을 뺀 값, 즉 100을 표현할 수 있는 최소의 바이트인 1바이트가 된다. 정규화 작업 두 번째에 따라 작은 경계값이 0이므로 원래의 값에서 0을 빼도 정규화한 값은 '0, 11'이 된다. 이를 각기 1바이트의 크기로 할당하여 '0, 11'의 키정보리스트로 표현된다.

4. MB-트리의 성능 분석

입력 작업에 대한 쓰기 비용은 인덱스가 써야하는 페이지 개수로부터 계산할 수 있고 검색 작업에 대한 읽기 비용은 인덱스가 읽어야 하는 페이지의 개수로 계산할 수 있다. 앞으로 이 두 개의 비용을 편의상 입력 작업 비용과 검색 작업 비용이라 칭한다. 이번 장에서는 이러한 입력 작업 비용과 검색 작업 비용을 B-트리, BFTL과 이론적으로 비교해 본다. 비교를 위해 B-트리 노드, BFTL의 노드, MB-트리의 중간 노드, 리프페이지를 1페이지로 구성한다.

B-트리는 최악의 경우 분할이 발생하기 전까지 한 엔트리에 대한 입력 작업을 수행하기 위해 한 리프노드를 써야 한다. 즉, 한 엔트리 당 1페이지를 쓰게 된다. 그에 비해 MB-트리는 출력버퍼가 최악의 경우로 구성될 때에도 최소 두개 이상의 엔트리 당 1페이지만 쓸 수 있으며 그 이유는 다음과 같다.

일단, MB-트리의 출력버퍼가 최악의 경우로 구성되는 조건은 출력버퍼에 입력된 엔트리들이 모든 리프노드에 균등하게 속하여 출력 작업 두 번째 단계에서 모두 같은 크기의 묶음이 형성될 때이다. 이렇게 되면 가장 큰 묶음을 쓰더라도 균일하게 분산된 효과로 각 묶음의 크기가 작아지므로 많은 엔트리를 한꺼번에 쓸 수 없게 되기 때문이다.

1페이지에 포함되는 엔트리의 개수를 m, MB-트리의 리프노드의 크기를 l 리프페이지, MB-트리의 출력버퍼의 크기를 w 페이지, 현재까지 인덱스에 입력된 엔트리의 개수를 n이라 표현하면 다음과 같은 계산이 가능하다. MB-트리의 출력버퍼는 최악의 경우,

$$\frac{\text{출력버퍼에 포함되는 유닛의 개수}}{\text{현재의 리프노드의 개수}}$$

의 엔트리를 한꺼번에 쓰므로, $(w \times m) / \{n / (m \times l)\}$ 이 한꺼번에 쓸 수 있는 엔트리의 개수이다. 식을 정리하면,

$$\frac{wm^2l}{n} \tag{1}$$

와 같다. 즉 1페이지에 포함되는 엔트리의 개수 m을 고정시킬 때, 출력버퍼의 크기(w)와 리프노드의 크기(l)가 크면 클수록 출력버퍼가 한꺼번에 많은 엔트리를 쓰는 것이다.

MB-트리가 최악의 경우 두개 이상의 엔트리 당 1개의 페이지만 쓰기 위한 조건은 다음과 같다.

$$\frac{wm^2l}{n} \geq 2 \tag{2}$$

식 (2)를 만족시킬 수 있도록 w, l을 구성하면 MB-트

리의 입력 작업 비용은 항상 B트리보다 작게 된다.

다음으로 MB-트리의 검색 작업 비용을 B-트리와 BFTL과 비교해 보겠다. MB-트리는 리프노드의 크기가 B-트리에 비해 크므로 대체로 B-트리보다 작은 높이(height)를 가진다. 그러므로 중간 노드의 검색 작업 비용은 대체로 B-트리보다 작다.

MB-트리의 리프노드에서의 검색 작업 비용은 리프노드의 크기에 의해서 결정된다. 메타페이지의 순서정보리스트의 크기(리스트를 구성하는 순서정보의 개수)가 작을수록 키정보리스트의 크기를 크게 할 수 있으므로 검색 작업 비용이 작아진다. 순서정보리스트의 크기를 작게 하기 위해서는 리프노드에 포함되는 엔트리의 개수가 작아져야 하고 그에 따라 리프노드의 크기가 작아져야 한다.

하지만 MB-트리의 리프노드에서의 검색 작업 비용을 줄이기 위해 리프노드의 크기를 너무 작게 할 필요는 없다. MB-트리의 메타페이지의 순서정보리스트 크기가 키정보리스트의 크기보다 2배로 구성될 경우 MB-트리는 리프노드에서 최소의 검색 작업 비용을 가진다. 이렇게 되면 2개의 키정보로 구분되는 각각의 구간에 포함되는 순서정보의 개수가 1개가 되므로 1개의 순서정보에 해당하는 리프페이지만 읽으면 되기 때문이다. 리프노드에서의 전체 검색 작업 비용은 메타페이지를 읽는 비용까지 합쳐서 총 2페이지 읽기 비용이 된다. 이는 B-트리의 리프노드에서의 검색 작업 비용이 1페이지 읽기 비용이므로 1페이지 읽기 비용만큼 크다.

MB-트리는 위에서 언급한대로 리프노드의 크기를 적절히 구성하면 전체 검색 작업 비용을 대체로 B-트리와 비슷한 값으로 만들 수 있다. 중간 노드에서의 검색 작업 비용은 대체로 B-트리보다 작고 리프노드에서는 B-트리보다 1페이지 읽기 비용만큼 크기 때문이다.

반면 BFTL의 검색 작업 비용은 최악의 경우 B-트리보다 C배 만큼 크다. 최악의 경우 노드전환테이블의 모든 가상 노드가 C개의 조각으로 구성되는 list를 가지게 되고 검색 시 list를 따라 LBA 상의 페이지들을 C번만큼 탐색하게 된다. 노드마다 기존 B-트리에 비해 C배의 검색비용을 가지므로 전체 검색 비용도 B-트리의 C배가 된다.

정리해보면 MB-트리는 리프노드의 크기에 대한 적절한 설정으로 B-트리의 쓰기 비용을 줄이면서도 B-트리에 가까운 검색 성능을 보일 수 있다. 표 2에 이론적인 성능 분석 결과를 간략히 정리해 보았다.

표 2. MB-트리 성능 비교 분석

구분	B-트리	BFTL	MB-트리
입력 작업 쓰기 비용	대	소	소
검색 작업 읽기 비용	소	대	중

5. 결론

본 논문에서는 플래시 메모리에 B-트리를 효율적으로 적용하기 위한 MB-트리 구조를 제안하였다. 출력버퍼를 사용하여 입력 작업의 비용을 줄였고 메타페이지를 사용하여 검색 작업의 비용을 개선하여 BFTL에서 입력 작업

비용을 줄이기 위해 검색 작업 성능을 크게 희생시켰던 문제점을 해결했다. 또한 BFTL처럼 FTL위에 또 다른 계층을 이루지 않고 인덱스 구조 자체를 변경하였다.

이러한 결과로 MB-트리는 기존 B-트리의 수정 본으로서 기존 B-트리의 우수한 특징을 보존하면서 동시에 플래시 메모리상에서 입력 작업과 검색 작업 모두를 효율적으로 수행할 수 있는 구조를 가진다.

또한 MB-트리는 리프노드의 크기를 적절히 조절함으로써 입력 작업의 비용과 검색 작업의 비용을 상황에 따라 유연하게 조절할 수 있고 BFTL보다 두 비용을 효율적으로 trade-off할 수 있는 장점을 가지고 있다. 이는 쓰기 비용에 매우 민감한 시스템에서나 검색 비용에 매우 민감한 시스템에서도 인덱스 구조의 변경 없이 리프노드 크기의 조절만으로 유연하게 인덱스를 적용시킬 수 있는 장점을 가지는 것이다.

앞으로는 이 논문에서 제시한 아이디어와 이론적인 비용 분석을 검증하기 위해 기존 B-트리와 BFTL 그리고 MB-트리를 구현하여 다양한 조건에서 실험을 수행할 예정이다. 그 후에 MB-트리를 이용한 플래시 메모리상의 다른 연구 이슈들에 대해 연구해 나가려고 한다.

참고 문헌

- [1] Intel Corporation, "Understanding the Flash Translation Layer(FTL) Specification".
- [2] Intel Corporation, "Software Concerns of Implementing a Resident Flash Disk".
- [3] Intel Corporation, "FTL Logger Exchanging Data with FTL Systems".
- [4] A. Kawaguchi, S. Nishioka, and H. Motoda, "A Flash-Memory Based File System," USENIX Technical Conference on Unix and Advanced Computing Systems, 1995 .
- [5] K. Han-Joon, and L. Sang-goo, A New Flash Memory Management for Flash Storage System, Proceedings of the Computer Software and Applications Conference, 1999.
- [6] C.H. Wu, L.P. Chang, and T. Kuo, "An Efficient B-Tree Layer Flash-Memory Storage Systems," Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA 2003), 2003.
- [7] M.J. Folk, B. Zoellick, and G. Riccardi, File Structure An Object-Oriented Approach with C++, Addison-Wesley, 1998.
- [8] E. Gal, and S. Toledo, "Algorithms and Data Structures for Flash Memories," ACM Computing Surveys, pp. 138-163, 2005.
- [9] R. Bayer, and E. M. McCreight, "Organization and Maintenance of Large Ordered Indices," Acta Informatica, 1972.